Improving Disk I/O Performance on Linux

Carl Henrik Lunde, Håvard Espeland, Håkon Stensland, Andreas Petlund, Pål Halvorsen
University of Oslo and Simula Research Laboratory
Norway
{chlunde, haavares, haakonks, apetlund, paalh}@ifi.uio.no

Abstract

The existing Linux disk schedulers are in general efficient, but we have identified two scenarios where we have observed a non-optimal behavior. The first is when an application requires a fixed bandwidth, and the second is when an operation performs a file tree traversal. In this paper, we address both these scenarios and propose solutions which both increase performance.

1 Introduction

Disk scheduling has been a hot topic for decades, and numerous algorithms have been proposed [1]. Many of these ideas have been imported into commodity operating systems, and the Linux deadline I/O, Anticipatory and completely fair queuing (CFQ) schedulers are generally efficient for many scenarios. There are, however, two scenarios where the existing mechanisms show severe weaknesses, and we here propose two small enhancements addressing these types of operation.

The CFQ algorithm allocates the same amount of input/output (I/O) *time* for all queued processes with the same priority. When requesting data from a disk, this can lead to differences in throughput between processes, depending on how much disk search that happens within its timeslice and the placement on the disk. To improve support for real-time processes requiring a fixed bandwidth, we have implemented a new priority class with quality of service (QoS) support for bandwidth and deadline requirements in CFQ. This new class provides enhanced real-time support while improving the overall performance compared to the existing classes. As an example, experiments show that the new bandwidth-based queue was able to serve 24 DVD-quality video streams while the existing CFQ-RT queue managed to serve 19 streams without deadline misses.

Furthermore, current in-kernel disk schedulers fail to optimize sequential multi-file operations like traversing a large file tree. This is because the application only reads one file at a time before processing it. We have investigated a user-level, I/O request sorting approach to reduce inter-file disk arm movements. This is achieved by allowing applications to utilize the placement of inodes and disk blocks to make a one sweep schedule for all file I/Os requested by a process, i.e., data placement information is read first before issuing the low-level I/O requests to the storage system. Our experiments with a modified version of the `tar` archiving utility show reduced disk arm movements and large performance improvements. As an example, a `tar` of the Linux kernel tree was 82.5 seconds using GNU `tar`, while our modified `tar` completed in 17.9 seconds.

This paper describes our enhancements and present experiments which demonstrate the performance gains. However, many results and discussions are omitted from this paper, but more details and results are presented in [2].

2 Adding QoS to CFQ

Completely Fair Queuing (CFQ) is the default I/O scheduler on most GNU/Linux distributions. Fairness among I/O requests is provided on a time basis instead of throughput, which means that processes doing

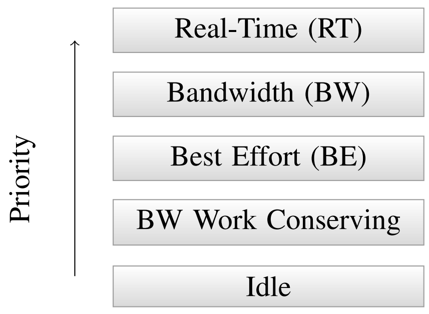


Figure 1: Priority levels in modified CFQ

random access gets lower throughput than a process with sequential I/O. CFQ does this by maintaining a number of queues per process, and each process is served periodically. The active process gets exclusive access to the underlying block device for the duration of the time slice, unless preempted. The length of the time slice is calculated by the priority level and the number of others processes waiting for requests. The issuing process belong to one the following priority classes: Realtime, Best Effort, or Idle.

To better support processes that require sustained throughput, e.g. video streaming, we propose adding a bandwidth class to CFQ which will provide fairness on throughput instead of I/O time. By adding this as a priority class in CFQ instead of a separate scheduler, we encourage coexistence with other I/O users in an attempt to be as non-intrusive as possible.

2.1 Bandwidth class priority

The main problem with existing solutions is that they are often special purpose schedulers suited for one specific task. For server consolidation one may want to mix proportional share with fixed rate and deadline requirements. However, there are no available solutions included in Linux 2.6.29. Several projects working on similar problems, but they are mostly for virtualization solutions, and instead of guaranteeing at least a certain bandwidth they throttle to a bandwidth without any work-conservation. This means that we would have to configure throttling of *all* processes if we wanted to guarantee a bandwidth for a single process. We do not consider this desirable on a multimedia/content server.

With this in mind, we have designed a new QoS class for the CFQ I/O Scheduler, which we call BW as shown in figure 1. It will reserve bandwidth in terms of bytes per second in contrast to proportional disk time. The solution has the following design requirements and properties: The solution should be non-intrusive, i.e., if there are no active BW class streams, the system should work as it does today. An application must be able to request a specific bandwidth, and this bandwidth should be delivered to the application at the expense of any BE-class readers. The bandwidth should be measured by throughput (as bytes per second), not with *disk time* and not proportional to load. To control buffer requirements in the client, an individual process may specify a deadline for its requests. The deadlines are soft for two reasons: We do not implement proper admission control in the I/O scheduler, and we do not estimate the cost of I/O operations. To the extent allowed by the request deadlines and other requirements of the system, the scheduler should optimize the requests for higher global throughput by using an *elevator*. Any reserved bandwidth which is unused should be redistributed fairly amongst BE class readers, because it is important to support work-conservation to maximize global throughput. If no BE-class readers have requests queued, the BW-class readers who have exceeded their reserved bandwidth, but have pending requests, should be able to consume all the available bandwidth. There are a number of reasons why a stream may reserve more bandwidth than it requests, for example VBR video and buffering. Conversely, VBR and caching might mean that the consumed bandwidth is less than the reserved bandwidth. For simplicity, we only enter work-conservation in the BW class when there are no BE class readers with requests pending. Another approach might treat a BW reader as a BE reader when they have exceeded the reserved bandwidth. However, we have not defined a policy for fair sharing of the extra bandwidth between BW class readers in the current design.

A key component for getting a QoS system to work is admission control, a component which receives the QoS-requirements from an application (and characteristics about the round time, which file is going to be read, etc.). The admission control then judges whether the I/O scheduler will be able to serve the stream given the current load and the stream characteristics. If we grant all requests for bandwidth and low latency, the promises will be broken. Because we allow the RT class to preempt BW queues, an admission control system must also control RT class streams. The current implementation does not yet have a system for admission control, and we think that this should be implemented outside the kernel I/O scheduler.

2.2 Implementation

We manage bandwidth using token buckets which is a normal strategy for QoS and data rate control in networking systems. It has also been used in other disk scheduling systems like the APEX I/O Scheduler [4]. A token bucket allows bursts, which are needed for good performance in a streaming scenario. Even though bursts are allowed, we get good control over average bandwidth.

Management of the tokens is fairly simple: When a request is dispatched, we remove the same number of tokens as the size of the request, converted to bytes. When we want to check if there are tokens available in the bucket, e.g., before dispatching a request, we must first check if any time has passed since we last updated the bucket. The new number of tokens is calculated as:

$$\texttt{bw\_tokens} = \texttt{bw\_tokens} + \frac{\texttt{bw\_iorate} \cdot (\texttt{jiffies} - \texttt{bw\_last\_update})}{\texttt{HZ}}$$

If the number of tokens is larger than the bucket size, we set the number of tokens equal to the bucket size. Furthermore, if we are in work-conservation mode, the number of tokens may be negative. Consider a case where a process has been allowed to dispatch 40 MiB of I/O in one second, because no one else used the system. If the `iorate` of the process was only 1 MiB/s this would mean that the bucket would have -39 MiB tokens. If the system then became 100% utilized with best effort processes we would not be allowed to schedule new requests for a period of 39 seconds. While this is fair regarding the average bandwidth (1 MiB/s) it would often be undesirable and could be considered unfair. To limit this effect, we only allow the bucket to have $\frac{\texttt{bucket\_size}}{2}$ negative tokens after work-conservation. By allowing the user to adjust the limit, this effect can be tuned, or completely disabled.

2.3 Evaluation

We have done a performance evaluation of the new BW class compared to the existing solutions available in Linux 2.6.29. To create a realistic multimedia scenario, we consider a server that provides streaming video to multiple clients. High definition video (1080p) on the most popular high definition storage format today (Blu-ray) has a theoretical maximum bitrate of 48 Mbit/s. Our experience indicate that the average bitrate for most high definition movies are around 25 Mbit/s ($\sim$3 MiB/s), so we have chosen this bitrate for our video streams. To make sure the scheduler can handle different bandwidths, we added another class of video streams, CBR video at 1 MiB/s, which is at the upper side of DVD bandwidth.

In this section, the word “greedy reader” is used for a process that consumes data at a rate faster than the disk can provide, for example the program `md5sum` consumed 325 MiB/s on our system, while the data rate of the disk is around 50-110 MiB/s.

Performance Limitations and Need for QoS

In the first scenario, we wanted to find the maximum number of reserved readers that we could add while still maintaining deadlines. Each reserved reader had a deadline of one second for each request, so the scheduler had to serve each stream once per second.

We also wanted to evaluate how the priority methods available in the scheduler would allow us to run other jobs concurrently at best effort, so we also added three processes doing random reads of 4 KiB, at most 20 times per second. Another set of best effort readers was added to see how streaming processes would impact the system performance and reserved readers, so we added five processes doing streaming at up to 4 MiB/s with a block size of 64 KiB. Note that we do not evaluate the performance of these streams: we are pushing the reservations to the maximum, with the consequence that they will be starved. If this is considered a problem, the admission control system must deny reservations before this happens.

Streams		CFQ			AS	Deadline	noop
1 MiB/s	3 MiB/s	BW SSF	BW C-SCAN	RT			
16	10	0	0	0	479	345	281
17	10	0	0	0	482	330	398
18	10	0	0	0	492	300	783
19	10	0	0	0	512	276	1061
20	10	0	0	179	505	288	1123
21	10	0	0	186	531	381	1075
22	10	0	0	276	517	947	1061
23	10	0	0	N/A	549	1233	1064
24	10	0	0	N/A	553	1276	1054
25	10	182	188	N/A	536	1279	1033

Table 1: Deadline misses with 1 s deadline / round

The results are shown in table 1 for the bandwidth class tested with both shortest seek first (SSF) and C-SCAN elevator. We started with 10 1080p-streams and 16 DVD streams, which all the CFQ-based solutions could handle, but the lack of any QoS in AS, Deadline and NOOP meant that they were unable to handle the video streams any differently from the others, so the deadlines are missed. This shows that a priority mechanism *is needed and useful* in this scenario.

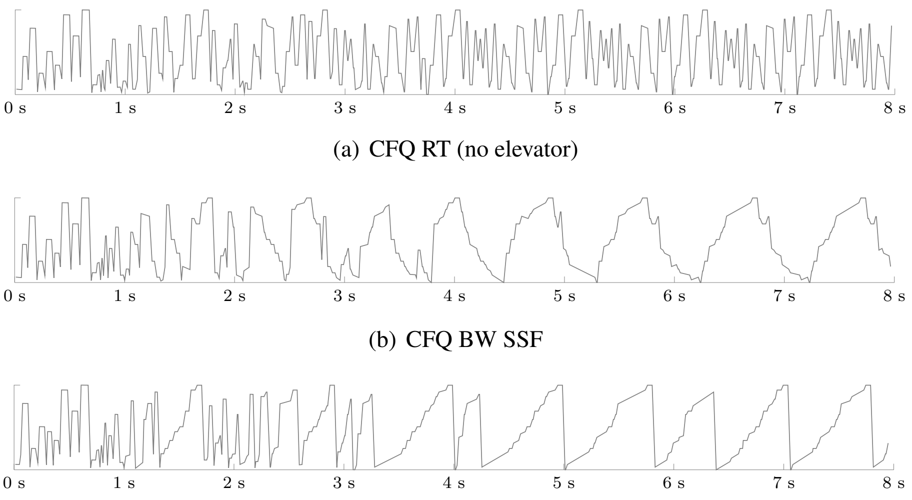
As we increased the number of DVD streams to 20, the RT class in CFQ also failed to maintain the deadlines. We contribute this mainly to the fact that the RT class serves the streams in a FIFO-like order, without any high-level elevator. We can see that the BW class we created, which does reorder streams, can handle 5 more streams. In figure 2, we plot the output of `blktrace` to show how the elevators affect the disk head position, i.e., the disk head movement according to seeks. The conclusion is that it is worth adding re-ordering of requests between streams, which in this case allowed us to support 17% more streams without deadline misses. The design goal of achieving higher global throughput has therefore been achieved.

When the number of streams was increased to 33, we discovered a new problem with the RT class: one stream was completely starved (no I/O). This is the reason for "N/A" in the table. Similar behaviour occurs for BW but at a later time. However, proper admission control should never allow this to happen.

Isolation

There are two types of isolation we want from our system: Best effort readers should not affect reserved readers, and greedy reserved readers should not get more bandwidth than reserved if it would affect any best effort reader.

The first experiment to see if this goal has been reached, is a simple benchmark based on the previous section: we remove all best effort readers from the first scenario with deadline misses (35 reserved readers). If the number of deadline misses is consistently less, we know the goal has *not* been reached. Our experiments show little change in deadline misses when we remove all the best effort readers, meaning that reserved readers are isolated from the best effort readers.



(a) CFQ RT (no elevator)

(b) CFQ BW SSF

(c) CFQ BW C-SCAN

Figure 2: Disk head movements with 34 media streams

A greedy BW class stream should not get more bandwidth than reserved if it would affect any best effort reader. To measure this, we have created a scenario with two greedy streams: one best effort and another reserved reader at 3 MiB/s. The initial bucket size for the reserved reader was 3 MiB. The result was that the reserved reader got 3.08 MiB/s, and the best effort stream achieved a rate of 100.1 MiB/s. This means that the BW stream class got slightly higher bandwidth than it theoretically should, but within reasonable limits. Since the main goal of limiting the bandwidth of the process is to prevent denial of service and starvation, as would happen if this was an RT class stream, we can conclude that isolation of the BW class behaves as expected.

Further evaluation of isolation issues can be found in [2].

Work-conservation

There are two work-conservation scenarios we must evaluate when a reservation reader has requested more bandwidth than it uses (over-provisioning). In both scenarios, we must have at least one greedy reader that reaches 100% utilization. We have used two greedy readers in order to check for fairness as well. In the first scenario, the greedy readers are in the best effort class with the default priority level (4), and in the second scenario, they have a reserved bandwidth of 4 MiB/s and a scheduling deadline of 500 ms. To measure global throughput without our bandwidth class, we have a baseline with only best effort readers as listed in table 2, showing that the video stream receives the requested bandwidth, and the greedy readers share the remaining bandwidth. In both experiments, the over-provisioning BW class reader has reserved 32 MiB/s, but it will only consume 8 MiB/s. It performs reads four times per second (2 MiB blocks) and has requested a 200 ms scheduling deadline.

Stream	Reserved	Requested	Result	Latency
Video	BE	8 MiB/s	8.0 MiB/s	313 ms max
Greedy A	BE	∞	26.5 MiB/s	253 ms max
Greedy B	BE	∞	26.5 MiB/s	243 ms max

Table 2: Baseline for over-provisioning evaluation

The results in table 3 and 4 both show aggregate bandwidth equal to or slightly above the baseline in table 2, i.e., utilization around 100%. The conclusion is that work-conservation works even with over-provisioning, otherwise a drop in aggregate bandwidth should be visible. In this case, we reserved 24

Stream	Reserved	Requested	Result	Latency
Video	32 MiB/s	8 MiB/s	8.0 MiB/s	190 ms max (42 ms avg)
Greedy A	BE	∞	25.5 MiB/s	779 ms max
Greedy B	BE	∞	30.5 MiB/s	700 ms max

Table 3: Over-provisioning has little effect on global throughput with best effort readers

Stream	Reserved	Requested	Result	Latency
Video	32 MiB/s	8 MiB/s	8.0 MiB/s	179 ms max (60 ms avg)
Greedy A	4 MiB/s	∞	4.1 MiB/s	787 ms max (487 ms avg)
Greedy B	4 MiB/s	∞	55.2 MiB/s	128 ms max (36 ms avg)

Table 4: Over-provisioning for BW class

MiB/s more than we used. Fairness amongst the best effort readers in table 3 is slightly skewed. This may be an indication that the preemption algorithm does not give the preempted task sufficiently extra time on the next schedule. When a process is allowed to be greedy beyond reservation, our design stops considering deadlines in the normal way. The deadline is set to the time when a claim for this bandwidth would be legitimate, i.e., when the number of tokens in the bucket is positive. This is why the deadlines for “Greedy A” in table 4 may seem broken, but because it has received more bandwidth than it requested, its buffers should cover the extra latency. We do, however, limit how many negative tokens we count, in order to limit the worst case scheduling delay in such a scenario.

We also consider another scenario, where we have best effort readers with limited bandwidth usage, and one or more greedy readers in the BW class. The setup used is similar to the last experiment, but limited the best effort readers to 8 MiB/s. The reserved reader still reserves 32 MiB/s, but will use anything it can get. The result should be that both best effort readers should get 8 MiB/s, because we know they *can* get that if the BW class was throttled at 32 MiB/s.

Stream	Reserved	Requested	Result	Latency
Greedy	32 MiB/s	∞	63.5 MiB/s	138 ms max (31 ms avg)
Video A	BE	8 MiB/s	8.0 MiB/s	676 ms max (48 ms avg)
Video B	BE	8 MiB/s	8.0 MiB/s	739 ms max (65 ms avg)

Table 5: Work-conservation for BW class

Table 5 shows that work-conservation works, the aggregate throughput is well above the baseline. The other important goal here is that the work-conservation does not affect the performance of the best effort readers.

3 Userspace I/O scheduling for multi-file operations

The in-kernel disk schedulers in current Linux distributions provide efficient means to optimize the order of issued, in-queue I/O requests. However disk schedulers in general fail to optimize sequential multi-file operations, like the operation of traversing a large file tree, because only the requests from a single file are available in the scheduling queue at a time, i.e., giving possibly large inter-file seeks on a rotational disk.

Our proposed approach for this problem is to use a two-pass scheduler in userspace which first retrieve metadata information about block placement and then perform file I/O operations based on data

```
def archive(path):
    next = path
    do
        stat next
        if is_directory:
            add to archive(next)
            for file in next:
                inode_cscan_queue.add(next)
        else:
            FIEMAP next
            block_sort_queue.add(next)
    while (next = inode_cscan_queue.next())

    flush()

def flush():
    for file in block_sort_queue:
        add to archive
```

Listing 1: Modified `tar` traversal algorithm

location on disk minimizing inter-file disk seeks. An approach sorting data first according to metadata has been proposed for the `readdir` system call on the Linux kernel mailing list (LKML) in [3], and we use the same general idea giving the two following phases:

1. The file entries can be sorted by either inode number, which one assumes is correlated with the physical location on disk, or better by logical block address. Such information is for example available in ext4 and xfs through the `FIEMAP` ioctl, and the `FIBMAP` ioctl with superuser privileges on ext3. The first pass then reads data placement information in C-SCAN order which is used as input to the second phase.

2. Based on the metadata information retrieved in pass one, the second pass fetches file data in the order of the first data block of each file. Thus, the metadata is used to sort the files in a one-sweep order based on its first logical block number greatly reducing the inter-file seeks.

Our scheduler is placed in userspace for two main reasons. First, it can better be adapted to the application's access pattern better known by the application programmer. The second is that the kernel scheduler is not able to fully optimize requests when traversing a directory tree because programs usually reads a limited number of files at a given time. Enqueuing all files in a directory tree to the kernel I/O scheduler would require a deep queue to hold buffers for all the files. The normal kernel scheduler behavior of assigning deadlines to prevent starvation is incompatible with our goal of minimizing the total wall clock time of tree traversal. We therefore propose to perform such I/O scheduling in userspace to minimize the negative impact of this change.

3.1 Implementation case study: Optimizing the `tar` archive program

We demonstrate our approach by using a case study example: `tar` is an old Unix archiving utility that has been around for decades and is found on most systems. Its operation is to store (or extract) files from an archive file, i.e., `tar` traverses and a archives full directory tree and is therefore well suited as an example for our technique. We do not aim to implement a complete version with all the functionality of `tar`, just the standard function of creating a new archive of all the contents in a directory tree. The algorithm of our implementation is shown in listing 1 where we first traverse the directory tree and then read all the files. The directory tree is traversed in inode order, and we do not consider the location of directory data blocks. This is because it is currently not possible to use `FIEMAP` on directories. It also keeps the implementation simple, and we believe that the relative gain is low (there are generally more files than directories). Because new inodes are discovered during traversal, we cannot pre-sort the list. Therefore, we use a C-SCAN elevator that stores all unchecked paths sorted by inode number. The inode number is a part of the information from the `readdir` function. The file list is a sorted list, ordered by block number, and the block number is retrieved by using `FIEMAP` on the files. Because the block queue does not change while traversing it, a simple sorted list is used. The cost of our approach is that the application must allocate memory to hold the sorted file list. We store a couple of hundred bytes for each file, such as file name, parent directory, inode and block number. However, if an upper bound of memory is wanted, the `flush()` method may be called at any time, e.g., for every 1000 files or when N bytes RAM have been used. In our tests we did not use any limit on memory, and around 6 MiB of memory is required used for the the Linux source directory tree with about 22 500 directory entries.

3.2 Evaluation

By running our new version on a large directory tree (the Linux kernel source consisting of about 22 500 files) on the ext4 file system, we get an average runtime of 17.98 seconds for five runs on an aged file system. In comparison, the the original GNU `tar` version completes in 82.6 seconds. Another interesting experiment is how the performance improvement changes as the file system ages. To test this, we ran our improved `tar` and GNU `tar` on a file system which is aged by incrementally checking out new versions of the Linux source tree. In figure 3, we show the average runtime for 5 runs. The amount of change in each in step in the aging procedure is different, which is why step 1, 2, 10 and 17 are especially steep for GNU `tar`. What is interesting to see in this graph, is that the age (fragmentation) has much less impact on our implementation where the improvement factor increases from 3.95 to 4.75 as the file system ages.

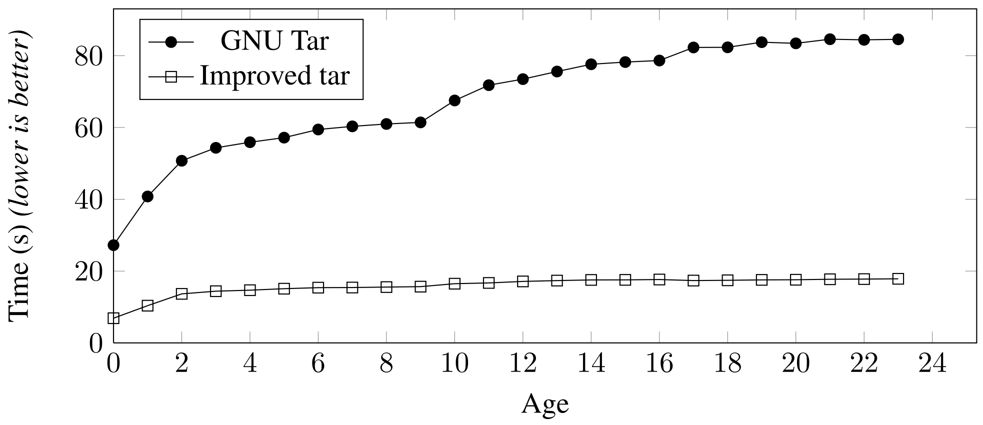


Figure 3: Traditional and modified `tar` performance as the file system ages

3.3 Applicability

The example above demonstrate the usability of our approach, and `tar` is only one example. For example, table 6 lists many good candidates for optimization through user space scheduling because of an almost identically behavior: 1) they must read all data and metadata for a full directory tree; 2) most of the operations are known in advance by the application programmer; and 3) they operate on a typically large number of files. A few of the programs listed as good candidates, such as `ls -l` and `du`, only need to read metadata, which the ext4 file system has improved with inode readahead, but some improvement should still be possible. It is also important that such core utilities have a low overhead, because they are often critical during recovery of overloaded systems.

Good candidates	**Bad candidates**
`cp -r`	`Firefox`
`zip`	`vim`
`rsync`	`ps`
`tar`	`cat`
`scp -r`	`man`
`rm -r`	`locate`
`find`	`less`
`mv`	
`du`	
`Tracker`	
`dpkg (database)`	
`ls -l`	

Table 6: Common Linux Software

The potential improvement in execution time will generally depend on how many concurrent I/O operations that can be reordered. Thus, the average gain may be low in some scenarios. This can for example be an issue for applications depending on external input, i.e., interactive applications and databases where there is no way to know what data to read until a user requests it. Similar issues exist for applications with data dependencies where one I/O operation decides which operation to do next, or where the data must be written to disk sequentially for consistency reasons. As an example of the former case, consider a binary search in a database index. Other examples of dependencies are that we need to read a directory block to find a file's inode number, and to read the file inode before we can read the file data. In such scenarios, the performance gain of our approach will be limited. Furthermore, modern Linux file systems store each file efficiently, e.g. by using extents. For large files, we therefore believe that there is not enough fragmentation to warrant reordering of requests within the same file. If our modification is applied in such a scenario, it will likely require intrusive changes in the program flow, and yield relatively low performance gains. Another relevant issue is whether it is likely that files are cached, i.e., if they have been recently used. A compiler such as `gcc` or rather the `cpp` preprocessor may seem like a good candidate, because it reads many small header files. In most workflows, however, header files will be cached reducing the potential performance improvement.

Our approach depend on retrieval of reliable file metadata. One problem with optimizing in the application layer is that we must make assumptions about the behavior in lower layers, such as file system and disk. The mapping between file placement and inode placement may not hold for other file systems, especially those with dynamic inode allocation, like *btrfs* [5]. Such a feature have also been discussed for ext4, but it will break compatibility, so it will likely be postponed until a new generation of the Linux extended file system is developed (i.e., ext5 or later). For such file systems, we do not know the effect of inode sorting, it might be less efficient than using the inodes in the order in which they are received from the file system. One solution to this problem would be to detect the file system type and disable sorting if it is an unknown file system, or on a file system where sorting is known to be less efficient.

4 Conclusion

In this paper, we have examined two I/O related issues on Linux. By implementing a new bandwidth priority class in CFQ, we were able to serve 17 % more video streams than the realtime class of CFQ. Furthermore, we demonstrated a technique for userspace scheduling of I/O requests when traversing directory trees, which retrieves disk block location information before issuing the I/O operations. The technique showed promising results with a speedup in a `tar` archive operation on the Linux kernel source from 82.5 seconds to 17.9 seconds.

References

[1] Pål Halvorsen, Carsten Griwodz, Vera Goebel, Ketil Lund, Thomas Plagemann, and Jonathan Walpole *Storage System Support for Continuous-Media Applications, Part 1: Requirements and Single-Disk Issues*, IEEE DSONLINE 5(1), 2004

[2] Carl Henrik Lunde *Improving Disk I/O Performance on Linux*, Master Thesis, 2009 *http://home.ifi.uio.no/paalh/students/CarlHenrikLunde.pdf*

[3] Theodore Ts'o *Re: [Bug 417] New: htree much slower than regular ext3*, LKML, 2003 *http://lkml.org/lkml/2003/3/7/348*

[4] Ketil Lund and Vera Goebel *Adaptive disk scheduling in a multimedia DBMS*, ACM MM, 2003 *http://doi.acm.org/10.1145/957013.957024*

[5] Btrfs Wiki *http://btrfs.wiki.kernel.org*